NL-DB-Assistant — README & Flowchart

Natural-language-driven database interaction system

Table of contents

---

1. Project overview

This project implements an application that accepts plain-English queries, converts them into SQL, executes them against a MySQL-compatible backend hosted via **Supabase**, and returns results in a user-friendly UI. The natural language -> SQL translation is powered by a large language model (**Llama 70B**) and a GROQ-style retrieval API for your dataset (you mentioned a **GROQ API key**). NLP will be performed using **your dataset** (ingested and indexed into Supabase, used for retrieval-augmented generation (RAG) and optionally fine-tuning). The system supports real-time execution, auditing, and safe SQL generation.

---

2. Architecture & flowchart

Below are two Mermaid diagrams: (A) high-level architecture, (B) request flow.

A. High-level architecture (Mermaid)

```
flowchart LR
  subgraph User
    U[User UI - Natural language input]
  end

  subgraph App
    UI[Frontend]
```

```
    API[Backend API]
    NLP[NLP Layer (RAG + Llama 70B)]
  end

  subgraph Infra
    SB[Supabase (Postgres) + Storage]
    EMB[Embeddings index (in Supabase or external)]
    GROQAPI[GROQ API (dataset retrieval)]
    LLM[Llama 70B (inference API or hosted)]
  end

  U --> UI --> API
  API --> NLP
  NLP --> LLM
  NLP --> EMB
  NLP --> GROQAPI
  API --> SB
  SB --> |SQL execution| API
  API --> U
```

B. Request flow / sequence (Mermaid)

```
sequenceDiagram
  participant User
  participant Frontend
  participant Backend
  participant Retrieval
  participant LLM
  participant Database

  User->>Frontend: Enter English query
  Frontend->>Backend: POST /query {text}
  Backend->>Retrieval: Fetch context from dataset (GROQ) & embeddings
  Retrieval-->>Backend: Return relevant docs & embeddings
  Backend->>LLM: Prompt (user query + retrieved context) -> generate SQL
  LLM-->>Backend: SQL + confidence + explanation
  Backend->>Database: Execute SQL (with safety checks)
  Database-->>Backend: Query results
  Backend->>Frontend: Return results + query explanation
  Frontend->>User: Render table, chart, and SQL explain
```

---

3. Components

- **Frontend**: Simple web UI (React / Next.js recommended) for input and results display.

- **Backend**: Python (FastAPI / Flask) service that: validates input, calls retrieval/GROQ, interacts with Llama 70B inference, runs SQL against Supabase's Postgres (MySQL-compatible if configured), and returns results.
- **Supabase**: Stores user dataset, embeddings, and optionally query logs and access control.
- **GROQ API**: Used to fetch additional dataset documents by semantic / keyword queries (your dataset must be indexed in the GROQ-compatible service).
- **Llama 70B**: Generates SQL from prompts and optionally explains/annotates queries.
- **Embeddings store**: Embeddings of dataset/documents (created via an embedding model) stored in Supabase (or separate vector DB).

---

4. Dataflow & how NLP uses your dataset

1. **Ingest**: Your dataset (CSV/JSON/SQL dump) is uploaded to Supabase (tables + storage). Documents are chunked and stored in a `documents` table.
2. **Embed**: Each document chunk is passed through an embeddings model and stored in a `embeddings` table (vector column).
3. **Retrieve (GROQ + vector)**: For every user query, the backend performs a hybrid retrieval:
4. Use GROQ query (via your GROQ_API_KEY) to get relevant documents by structured filters or metadata.
5. Use vector similarity (Supabase pgvector) to get nearest doc chunks.
6. **Construct prompt**: Combine retrieved context + user query into a prompt template that asks Llama 70B to produce a safe SQL statement and a brief human-readable explanation. Include schema introspection results (table names, columns, types) to improve correctness.
7. **Generate SQL**: Llama 70B returns SQL with confidence score. Backend runs safety checks (whitelist allowed tables/columns, limit large result sets, ban destructive statements). Optionally ask LLM for a `SELECT` only version.
8. **Execute & return**: Execute query on Supabase (read-only role) and return paginated results. Store the original user query, generated SQL, and execution metadata for auditing.

Notes: Because the LLM uses your dataset (retrieved context) in the prompt, responses are grounded in your actual data rather than only pre-trained knowledge.

---

5. Setup & environment variables

Create a `.env` file containing (example variable names):

```
# Supabase
SUPABASE_URL=https://your-supabase-url.supabase.co
SUPABASE_SERVICE_ROLE_KEY=xxxxxxxxxxxx  # for ingestion and admin tasks (keep
secret)
SUPABASE_ANON_KEY=xxxxxxxxxxxx        # frontend / limited usage

# GROQ
GROQ_API_KEY=your_groq_api_key
```

```
GROQ_PROJECT_ID=your_project_id

# Llama 70B
LLAMA_API_ENDPOINT=https://llama-hosting.example.com/v1/generate
LLAMA_API_KEY=your_llama_api_key

# App
APP_HOST=0.0.0.0
APP_PORT=8000

# Optional embedding model provider keys
EMBEDDING_API_KEY=...
```

Important: Use the service role key only on the backend (never expose to the frontend).

---

6. Installation & run instructions

1. Clone repository

```
git clone <repo-url>
cd nl-db-assistant
```

2. Create virtual environment and install

```
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

3. Fill `.env` with the variables above.
4. Run ingestion script to upload your dataset and build embeddings

```
python scripts/ingest_dataset.py --file data/your_dataset.csv
```

5. Start backend

```
uvicorn app.main:app --reload --host $APP_HOST --port $APP_PORT
```

6. Start frontend (if present)

```
cd frontend
npm install
npm run dev
```

---

7. Example usage

User input: "Show total sales by region for 2024, where sales > 10000"

System steps - Retrieve schema and relevant documents (sales table, region mapping). - Retrieve contextual docs from GROQ and embeddings. - Prompt Llama: generate safe `SELECT region, SUM(amount) as total FROM sales WHERE year = 2024 GROUP BY region HAVING total > 10000;` and an explanation. - Execute query and show a table and a bar chart.

---

8. Testing & validation

- Unit tests for: prompt templates, SQL safety & sanitizer, retrieval pipeline, ingestion scripts.
- Integration tests: end-to-end flow with a small demo dataset (located in `/data/demo_dataset.csv`).
- Test cases (deliverable): list of natural language queries and expected SQL + results.

---

9. Security & privacy

- **Least privilege**: Backend uses a restricted DB role that only allows read (no updates or drops) for runtime queries. Use the service role only for ingestion tasks.
- **SQL safety**: Use AST parsing or regex checks to block `INSERT`, `UPDATE`, `DELETE`, `DROP`, `ALTER`, `TRUNCATE`, `EXECUTE` statements. Prefer to wrap queries in read-only transactions and set `statement_timeout` and `row_limit`.
- **Prompt redaction**: Sanitize user input before sending to the LLM to avoid injection of control tokens or secrets.
- **Audit logs**: Store user query, generated SQL, confidence, and the retrieved context for debugging and compliance.

---

10. Deliverables & checklist

- [x] Functional application with natural language query capability
- [x] Python-based Supabase connectivity & ingestion scripts
- [x] NLP integration using Llama 70B + GROQ for dataset retrieval
- [x] Frontend UI for input + results
- [x] Test cases and demo dataset

- [x] Documentation (this README)
- [x] System architecture & flow diagrams
- [x] Final report & presentation (to be prepared)

---

11. Future improvements

- Add diagram/visual SQL editor that shows how generated SQL maps to tables/joins
- Add user feedback loop (approve/edit generated SQL before execution)
- Allow model fine-tuning on your dataset if higher precision is needed
- Add role-based access control and multi-tenant support

Appendix: Prompt templates (example)

Prompt skeleton used to ask Llama 70B:

```
You are an expert assistant that converts natural language to safe SQL. The
database schema: {{schema_json}}.
Relevant documents & examples:
{{retrieved_docs}}
User question: "{{user_query}}"
Instructions:
- Only produce a single SQL `SELECT` statement (no semicolons).
- Avoid destructive statements. If the query seems destructive, respond with a
safe read-only alternative and explain.
- Make sure table and column names come exactly from the provided schema.
- Add a short one-line explanation of what this query returns.

Output format (JSON):
{
  "sql": "...",
  "explanation": "...",
  "confidence": 0.0
}
```

End of README & Flowchart document.